Computer Architecture - HW 2

Connor Finley

September 24, 2017

1

Bits have no inherent meaning. Given the 32-bit pattern:

`1010 1101 0001 0000 0000 0000 0000 0010`

What does it represent, assuming it is ...

1.a

A 2's complement signed integer?

Negative binary; flip the bits and add one:

`1010 1101 0001 0000 0000 0000 0000 0010`

`0101 0010 1110 1111 1111 1111 1111 1101`

`0101 0010 1110 1111 1111 1111 1111 1110`

= -1,391,460,350

1.b

A MIPS instruction?

- Op code (first 6 bits): `1010 11`
  - Not 0, so not R-type instruction
  - Equal to `0x2B` => store word instruction (`sw`)
- `sw rt imm(rs)`
  - Format: `0x2B rs rt imm`
  - 5-bit `rs=01 000` = 8, $8, or $t0 (source)
  - 5-bit `rt=1 0000` = 16, $16, or $s0 (destination)
  - 16-bit `imm=0000 0000 0000 0010` = 2
- `sw $16 2($8)` or `sw $s0 2($t0)`
- Means that `MEM[$8+2] = $16` or `MEM[$t0+2] = $s0`

2

Determine the absolute value of a signed integer. Show the implementation of the following pseudo-instruction using three real instructions:

`abs $t1, $t2`

```
sra $t1, $t2, 31
xor $t2, $t2, $t1
subu $t1, $t2, $t1
```

3

For each pseudo-instruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may use the `$at` for some of the sequences. In the following table, `imm32` refers to a 32-bit constant.

3.a

`move $t1, $t2`

```
addu $t1, $zero, $t2
```

3.b

`clear $t5`

```
and $t5, $t5, $zero
```

3.c

`li $t5, imm32`

```
addiu $t5, $zero, imm32
```

3.d

`addi $t5, $t3, imm32`

```
addi $t0, $zero, 0xFFFF     # create mask
andi $t1, $t0, imm32        # get bottom bits using mask
xori $t2, $t1, imm32        # get top bits by filtering out bottom bits
or $t5, $t1, $t2            # combine into t5
add $t5, $t3, $t5           # add $t3
```

3.e

`beq $t5, imm32, Label`

```
addi $t0, $zero, 0xFFFF     # create mask
andi $t1, $t0, imm32        # get bottom bits using mask
xori $t2, $t1, imm32        # get top bits by filtering out bottom bits
or $t3, $t1, $t2            # combine into t3
beq $t5, $t3, Label
```

3.f

`ble $t5, $t3, Label`

```
slt $t0, $t3, $t5       # inverse of $t5 <= $t3, $t5 > $t3
beq $t0, $zero, Label   # branch if not $t5 > $t3
```

3.g

```
bgt $t5, $t3, Label
```

```
slt $t0, $t3, $t5        # $t5 > $t3
bne $t0, $zero, Label    # branch if $t5 > $t3
```

3.h

```
bge $t5, $t3, Label
```

```
slt $t0, $t5, $t3        # $t5 < $t3
beq $t0, $zero, Label    # branch if $t5 >= $t3
```

4

Translate the following statements into MIPS assembly language. Assume that `a`, `b`, `c`, and `d` are allocated in `$s0`, `$s1`, `$s2`, and `$s3`. All values are signed 32-bit integers.

4.a

```
if ((a > b) || (b > c)) {d = 1;}
```

```
slt $t0, $s1, $s0        # b < a
slt $t1, $s2, $s1        # c < b
or $t2, $t0, $t1         # (a > b) || (b > c)
bne $t2, $zero, set_d
j end

set_d: addi $s3, $zero, 1

end:
```

4.b

```
if ((a <= b) && (b > c)) {d = 1;}
```

```
slt $t0, $s1, $s0        # a > b
xori $t0, $t0, 1         # flip to find a <= b
slt $t1, $s2, $s1        # c < b
and $t2, $t0, $t1        # (a <= b) && (b > c)
bne $t2, $zero, set_d
j end

set_d: addi, $s3, $zero, 1

end:
```

5

Consider the following fragment of C code:

```
for (i=0; i<=100; i=i+1) { a[i] = b[i] + c; }
```

Assume that `a` and `b` are arrays of words and the base address of `a` is in `$a0` and the base address of `b` is in `$a1`. Register `$t0` is associated with variable `i` and register `$s0` with `c`. Write the code in MIPS.

```
addiu $t5, $zero, 100
L1:
    lw $t1, 0($a1)      # $t1 = b[i]
    addu $t2, $t1, $s0  # $t2 = b[i] + c
    sw $t2, 0($a0)      # a[i] = $t2
    addiu $a0, $a0, 4   # point to next a[i]
    addiu $a1, $a1, 4   # point to next b[i]
    addiu $t0, $t0, 1   # i += 1
    slt $t3, $t5, $t0   # $t3 = i > 100
    beq $t3, $zero, L1  # loop if (i <= 100)
```

6

Add comments to the following MIPS code and describe in one sentence what it computes. Assume that `$a0` is used for the input and initially contains `n`, a positive integer. Assume that `$v0` is used for the output.

```
begin:
    addi $t0, $zero, 0      # $t0 = 0
    addi $t1, $zero, 1      # t1 = 1
loop:
    slt $t2, $a0, $t1       # $t2 = n < $t1
    bne $t2, $zero, finish  # finish if n < $t1
    add $t0, $t0, $t1       # if input >= $t1, $t0 += $t1
    addi $t1, $t1, 2        # $t1 += 2
    j loop                  # repeat
finish:
    add $v0, $t0, $zero     # $v0 = $t0
```

The code produces the square number sequence, with each square outputted twice.

7

The following code fragment processes an array and produces two important values in registers `$v0` and `$v1`. Assume that the array consists of 5000 words indexed 0 through 4999, and its base address is stored in `$a0` and its size (5000) in `$a1`. Describe what this code does. Specifically, what will be returned in `$v0` and `$v1`?

```
add $a1, $a1, $a1
add $a1, $a1, $a1           # ... $a1 *= 4 === 20,000
add $v0, $zero, $zero       # $v0 = 0
add $t0, $zero, $zero       # $t0 = 0
outer:
    add $t4, $a0, $t0
    lw $t4, 0($t4)          # $t4 = $a0[$t0]
    add $t5, $zero, $zero   # $t5 = 0
    add $t1, $zero, $zero   # $t1 = 0
inner:
```

```
    add $t3, $a0, $t1
    lw $t3, 0($t3)          # $t3 = $a0[$t1]
    bne $t3, $t4, skip
    addi $t5, $t5, 1        # if $t3 == $t4: increment $t5
skip:
    addi $t1, $t1, 4
    bne $t1, $a1, inner     # point $t1 to next index while it's less than $a1
    slt $t2, $t5, $v0
    bne $t2, $zero, next    # if $t5 < $v0: next
    add $v0, $t5, $zero     # $v0 = $t5
    add $v1, $t4, $zero     # $v1 = $t4
next:
    addi $t0, $t0, 4        # point $t0 to next index
    bne $t0, $a1, outer     # break if $t0 < $a1
```

The loop finds the highest occurring element in the array (`$v1`), and keeps count of its occurrences (`$v0`).